Argonne NATIONAL LABORATORY

... for a brighter future

ALCF

Argonne Leadership Computing Facility

U.S. Department of Energy

UChicago ▶ Argonne LLC

Office of Science U.S. DEPARTMENT OF ENERGY

A U.S. Department of Energy laboratory managed by UChicago Argonne, LLC

BG/L Optimization Tips

Argonne Leadership Computing Facility

Dinesh Kaushik
Andrew Siegel
Argonne National Laboratory and University of Chicago

February 7, 2007

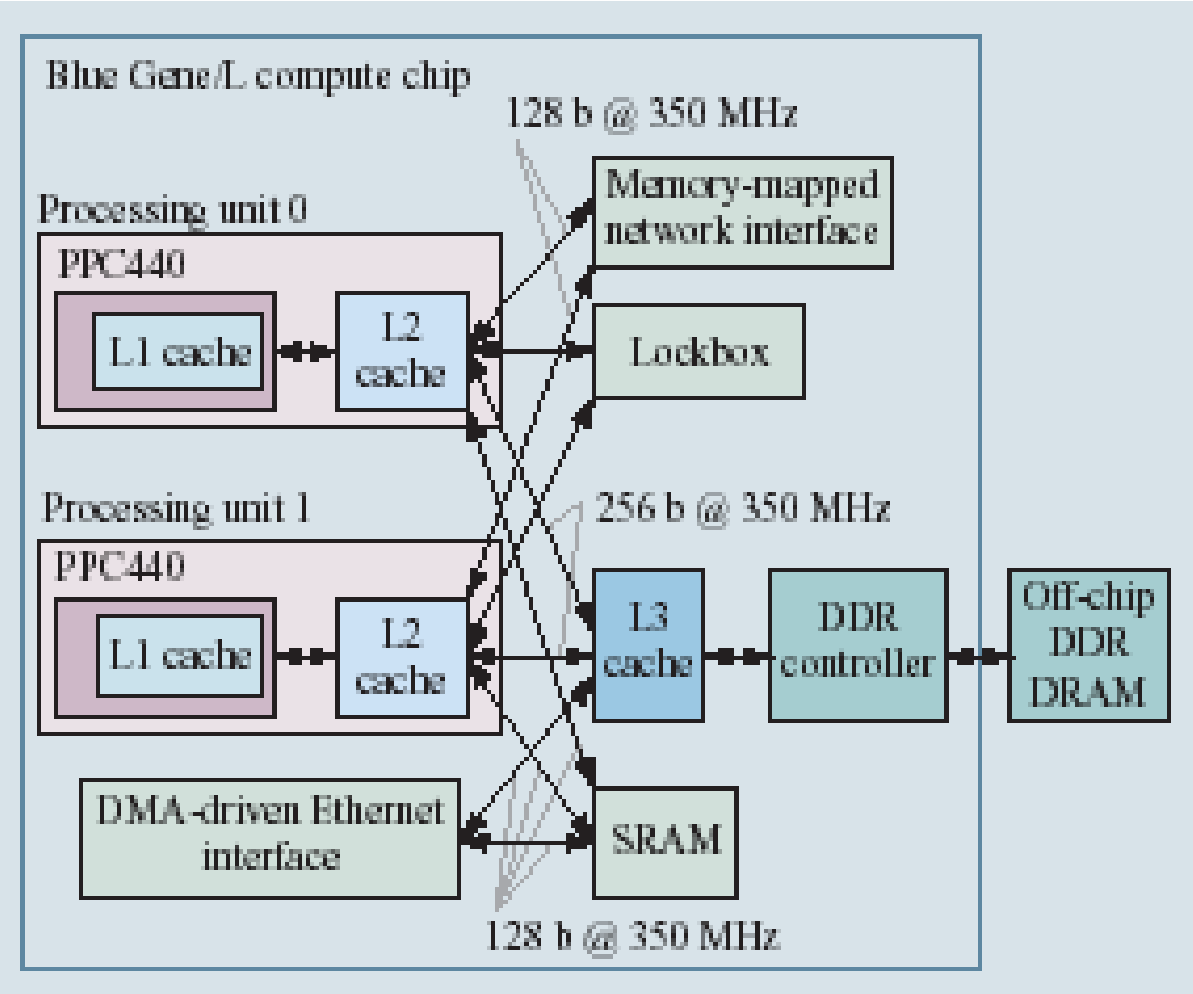
Practical Optimization Steps

- Start with those that require no code modification
  - Compiler switches
  - Virtual-node vs. co-processor mode
  - Using optimized libraries (DGEMM, MASSV, etc.)
  - Parallel opts
    - *MPI_EAGER_LIMIT*
    - *Explicit mapping*
    - *Etc.*
- Use directives within code
  - Alignment assertions
  - Aliasing assertions
  - Loop unrolling suggestions
  - Vectorization suggestions

Practical Optimization Steps

- Hierarchy of direct code modifications
  - Appropriate if performance bottlenecks are highly concentrated
  - Rearranging memory
    - *Cache reuse*
    - *Contiguous pairs of doubles allow for quad-word loads*
  - Use double-hummer intrinsics
    - *Register/instruction schedule still done by compiler*
  - Hand-Coding assembler

BG/L Compute Chip



Source: IBM

PPC440 Characteristics

- 32-bit architecture at 700 MHz
- Single integer unit
- Single load/store unit
- Special double floating-point unit (double hummer)
- Floating-point pipeline: 5 cycles
- Floating-point load-to-use latency: 4 cycles

Double FPU

- Double FPU has 32 primary floating-point registers, 32 secondary floating-point registers, and supports:
  - Standard PowerPC instructions, which execute on fpu0 (lfd, fadd, fmadd, fadds, fdiv, …), and
  - SIMD instructions for 64-bit floating-point numbers (lfpdx, fpadd, fpmadd, fpre, …)

Compute Chip Characteristics

- L1 Data cache
  - 32 KB total size, 32-Byte line size, 64-way associative, round-robin replacement
- L2 Data cache
  - Prefetch buffer, holds 16 128-byte lines
- L3 Data cache
  - 4 MB, ~35 cycles latency, on-chip
- Memory
  - 512 MB DDR at 350 MHz, ~85 cycles latency

Peak Flop/s

- 700 Hz * 2 flops/cycle * 2 fpus =
  - 2.8 GFlop/s theoretical peak per processor
- Assumes quite a few things
  - All FMAs
  - Perfect use of double hummer (more soon)
  - Significant cache reuse (e.g., not streaming)
  - Not load bound
  - Can fill 5-stage pipeline
  - Etc.
- Caution: %-peak is only meaningful in comparison to something.
  - 10% may be good, 1% may be good, 50% may be bad…

Memory Bandwidth

- L1-cache: can complete 1 quadword load per clock cycle:
  16B*700/s = 11.2GB/s

- Out of L1-cache: depends on complex three-level memory hierarchy
  Theoretical max = 3.7GB/s

IBM XL Compiler Optimizations

- General optimization levels
  - Default optimization = none (very slow)
  - -O: good place to start, use with -qmaxmem=64000
  - -O2: same as -O
  - -O3 -qstrict: can try more aggressive optimization but must strictly obey program semantics
  - -O3: aggressive, allows re-association, will replace division by multiplication with the inverse
  - -qhot: turns on high-order transformation module will add vector routines, unless -qhot=novector
  - -qreport: vectorization/optimization report on loops
  - -qipa: inter-procedure analysis; may cause very slow compilation

IBM XL Compiler Optimizations (cont.)

- Architecture flags
  - -qalign=… (fortran only)
  - -qarch=440 : generates standard powerpc instructions
  - -qarch=440d : will try to generate double FPU code
- Suggested steps on BG/L
  - -O -qarch=440 -qmaxmem=64000 (KB of memory used by compiler)
  - -O3 -qarch=440/440d (-qmaxmem=-1 is default at –O3)
  - -O4 -qarch=440d -qtune=440 (or -O5…)
  - -O4 = -O3 -qhot -qipa=level=1 -qarch=auto
  - -O5 = -O3 -qhot –qipa=level=2 -qarch=auto
- Use –v flag or check .lst file to see all flags used in compilation.

Compiler Listing

- -qsource –qlist
  - Creates .lst file containing assembler listing
  - Highly recommended when trying to squeeze performance out of numerical kernel
  - Try different compiler flags and study code that is generated to understand performance

Runtime Mode

- Virtual-node mode
  - Each processor on a node runs as its own MPI task and gets ½ total RAM (256MB each).
  - Use *cqsub -m vn*
- Co-processor mode
  - One CPU is used for message passing and the other for computation.
  - Compute processor gets full 512Mb RAM.
  - Use *cqsub –m co*

Optimized Libraries

- ESSL BG/L port available
- No plans for PESSL port
- Vanilla version of ESSL routines (BLAS, LAPACK, FFTW, etc.) performs poorly.
- See cheat sheet for more details/examples.

Cache Parity

- Memory errors occur at a small but nonzero rate
  - L1 date and instruction cache
  - TLB
- Usually correctable, but longer jobs are likely to see them more.
- Use Write Through policy
  - BGL_APP_L1_WRITE_THROUGH=1
- Bypass L1
  - BGL_APP_L1_SWOA=1
- Performance penalty ~ 10%-30%

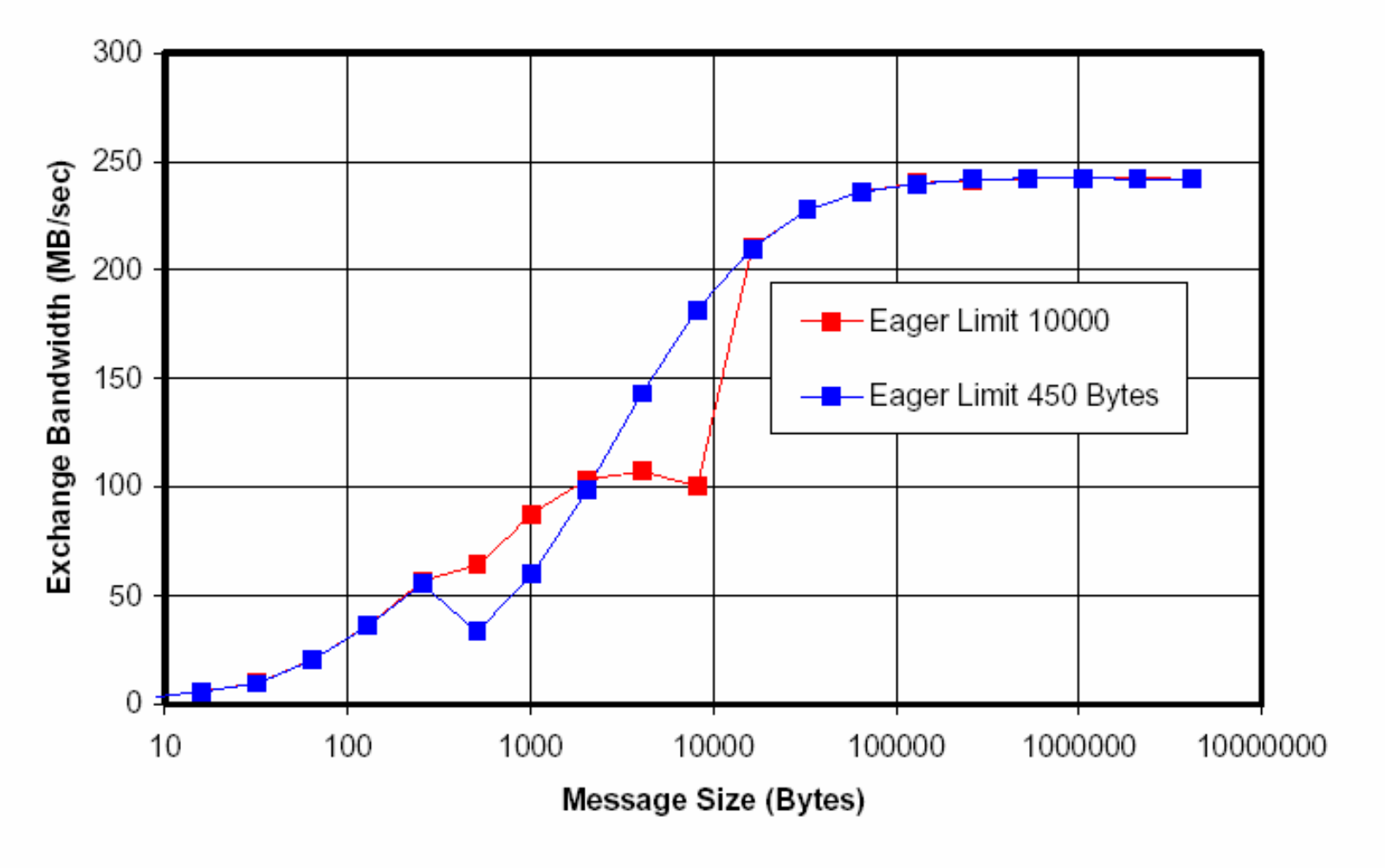
MPI Mapping

- With virtual node mode, experiment with BGLMPI_MAPPING=TXYZ.
  - This puts tasks 0 and 1 on the first node, tasks 2 and 3 on the next node, with nodes in x, y, z torus order.
  - The default layout is XYZT, which is often less efficient than TXYZ.
  - Also note that in TXYZ mode, you get two tasks per node if you have #tasks < 2*#nodes; otherwise, the XYZT layout will leave just one task on at least some nodes.
  - Can also write a mapfile to explicitly control processor mapping

EAGER_LIMIT

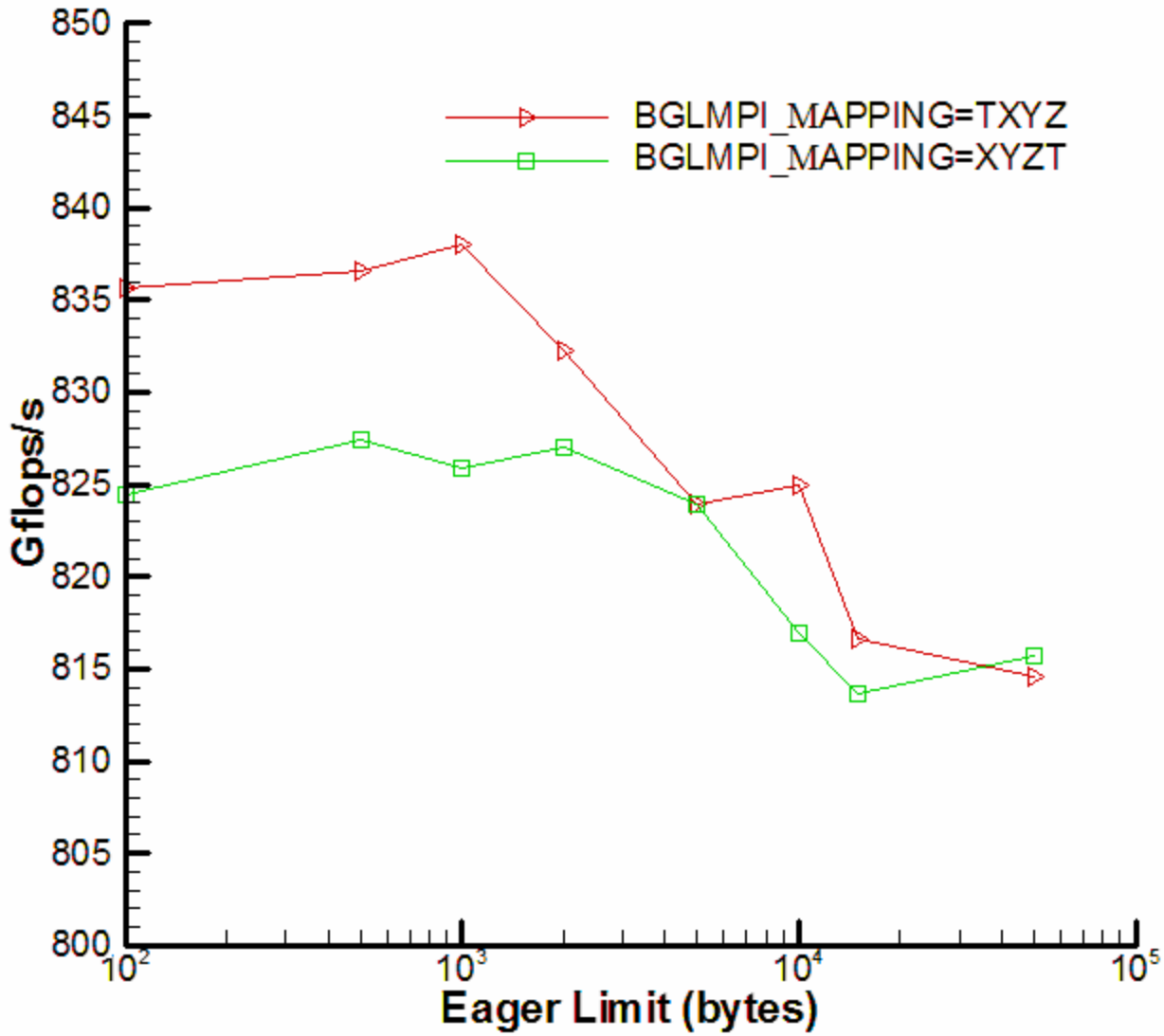
- BG/L can route messages either statically or dynamically.
- By default, small messages (those smaller than MPI_EAGER) are routed statically, and large ones are routed dynamically.
- These can be controlled with the following environment variables (see cheat sheet for passing these to Cobalt):
  - BGLMPI_EAGER = 1000 (default is 10000)
    - *Sets limiting message size in bytes for eager protocol*
  - BGLMPI_AE = 1
    - *To try adaptive route for eager message. Default is static.*

Random Exchange 8x8x8 Torus



Source: IBM

Mapping and Eager Limit on PETSc-FUN3D



Compiler Assertions

- Three compiler assertions are particularly important for generating optimal code:
  - Alignment
    - *call alignx(16,x(1)) Fortran*
    - *__alignx(16,x) C*
      - Inform compiler that variable x is aligned on a 16-byte boundary.
  - Aliasing
    - *#pragma disjoint(*a,*b) C only*
      - Inform compiler that a and b will not refer to overlapping memory.
  - Unrolling
    - *!ibm* unroll(n) Fortran*
    - *#pragma unroll(n) C*
      - Unroll inner loop that follows n elements.

Example with DAXPY

Fortran

```
call alignx(16,x(1))
call alignx(16,y(1))
!ibm* unroll(10)
do i = 1, n
y(i) = a*x(i) + y(i)
end do
```

C

```
double * x, * y;
#pragma disjoint (*x, *y)
__alignx(16,x);
__alignx(16,y);
#pragma unroll(10)
for (i=0; i<n; i++) y[i] = a*x[i] + y[i];
```

Annotations Example: STREAM triad.c

```
void triad(double *a, double *b, d
{
  int i;
  double ss = 1.2;
  /* --Align;;var:a,b,c;; */
  for (i=0; i<n; i++)
    a[i] = b[i] + ss*c[i];
 /* --end Align */
}
```

```
void triad(double *a, double *b, double *c, int n)
{
#pragma disjoint (*c,*a,*b)
  int i;
  double ss = 1.2;
  /* --Align;;var:a,b,c;; */
 if ( ((int)(a) | (int)(b) | (int)(c)) & 0xf == 0) {
   __alignx(16,a);
   __alignx(16,b);
   __alignx(16,c);
  for (_i=0;_i<n;_i++) {
    a[_i] = b[_i] + ss*c[_i];
 }
 }
 else {
    for (_i=0;_i<n;_i++) {
        a[_i]=b[_i]+ss*c[_i];
 }
  /* --end Align */
}
```

Performance of STREAM triad.c

Size	No Annotations (MB/s)	Annotations (MB/s)
10	1920.00	2424.24
100	3037.97	6299.21
1000	3341.22	8275.86
10000	1290.81	3717.88
50000	1291.52	3725.48
100000	1291.77	3727.21
500000	1291.81	1830.89
1000000	1282.12	1442.17
2000000	1282.92	1415.52
5000000	1290.81	1446.48

Double-Hummer Examples

- See ~siegela/examples/ on bgl
  - mxm
    - *In-cache matrix-matrix products using double-hummer intrinsics*
  - dotp
    - *dot product using double-hummer intrinsics and ensuring alignment*
  - ax+b

Listing File

- Use –qsource –qlist to generate friendly assembler listing.
- Good strategy is to tweak source, compiler options and diagnose with .lst output, rather than hand-coding assembler.

Performance Tools

- Currently installed performance tools
  - gprof for per-routine timings
  - memmon for detecting high-water memory mark
  - mpitrace for automatically timing mpi calls
  - stackmonitor for monitoring stack size
  - hpmlib preliminary port
  - papi for hardware counters
  - tau for more integrated and complex analysis
    - *Requires PAPI or hpmlib for hardware counters*
- See cheat sheet for examples of how to use.